NO CODE TEST RECORDING FOR iOS APPLICATIONS

A Thesis
Presented to
The Academic Faculty

by
Anushk Mittal

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science with the Research Option in the
School of Computer Science

Georgia Institute of Technology
July 2020

NO CODE TEST RECORDING FOR iOS APPLICATIONS

Approved by:

Dr. Alessandro Orso, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Qirun Zhang
School of Computer Science
Georgia Institute of Technology

Date Approved: July 27, 2020

ACKNOWLEDGMENTS

I wish to thank Dr. Alessandro Orso without whom none of this would have been possible.

I also wish to thank my mentor Dr. Shauvik Roy Choudhary, whom I worked under and with throughout my years of undergraduate research. He inspired me with the initial idea and helped develop the skills necessary to bring it to life. He guided me and supported me throughout the process, from ideation to completion.

I would also like to thank my parents, my professors throughout my undergraduate studies at Georgia Tech, and the College of Computing, who believed in me, supported me and guided me through the path of learning new things and venturing into the unknown.

If you want to change the world, you're at Georgia Tech! You can do that!

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

SUMMARY

Over the past decade, mobile apps have touched every sphere of life with ~4.5M applications available to download through Apple's App Store and Google's Play Store [1] that are expected to generate ~$1T in revenue by 2023 [2]. Today, an average American checks their phone once every 12 minutes [3], but testing these mobile apps is mostly unreliable and too resource expensive with current state-of-the-art technology. Specifically, testing iOS apps requires writing code using Xcode IDE that requires a development environment setup for all testers. These testers must also be familiar with coding for iOS apps as they need to interface with XCUITest API to write UI tests or verify the automatically generated code through Xcode's XCUITest Recorder. To address this issue and to make iOS testing accessible to everyone, we adopt the Barista technique [4] to passively record user interactions and build device-independent test scripts using any iOS device. We describe a three-part technique of recording user interactions through Objective-C swizzling, encoding generated test cases using a separately hosted server, and generating XCUITest files to run encoded test cases. We conclude with experimental results and discussions that demonstrate the effectiveness of our solution on a host of sample open-source applications that represent the most common and popular app categories and functionalities along with future directions on how this collected big data could be leveraged for intelligent insights. The goal of this research is to make testing approachable and easy for large corporations and indie developers alike. The presented tool has been made open-source at https://github.com/anushkmittal/iOSTestSDK.

CHAPTER I
INTRODUCTION

Over the past decade, mobile apps have touched every sphere of life from making new friends online to finding one's significant other, to banking, shopping and playing games. They have replaced physical objects on our desks, such as calculators, torches, notepads, stickies, and more. In 2019 alone, the App Store ecosystem supported $519 billion in billings and sales globally [5]. No doubt, mobile apps play a significant role in our lives and present a multi-billion dollar industry [6].

Like all traditional software, these apps must be tested to check for correct behavior under different input conditions and scenarios, especially when failures in an app can result in loss of reputation, and ultimately customers, for all online businesses.

These concerns are largely unanswered for iOS smartphones created and developed by Apple, Inc. iOS is the operating system that presently powers many of the company's mobile devices, including the iPhone, iPad, and iPod Touch. Initially unveiled in 2007 for the iPhone, iOS has been extended to support other Apple devices such as the iPod Touch (September 2007) and the iPad (January 2010). As of the 1st quarter of 2020, Apple's App Store contains more than 1.8 million iOS apps [1], and in 2019, apps available on the App Store were downloaded over 31 billion times [7]. Today, approximately 50% of all smartphone devices in the US are iPhones [8]. However, testing iOS apps is particularly challenging as major new iOS releases are announced yearly during Apple's Worldwide Developers Conference (WWDC) with support for the majority of iPhones and iPads. Between 2017 to 2020, in the last three years, Apple has released 10 new iPhones and 11 new iPad models. iPhones introduced new screens of sizes 5.8-inch,

6.5-inch, 6-inch while continuing required support for 5.5-inch, 4.7-inch, 4-inch, and 3.5-inch models. iPad models introduced 12.9-inch screens while continuing to support 11-inch, 10.5-inch, and 9.7-inch models. In this period, Apple also introduced multitasking support for iPad, which requires apps to respond to dynamic UI sizes. This makes testing app behavior for different configurations more challenging every year, and the problem compounds with the number of possible test device configuration parameters possible. Since the matrix of all device condition combinations is critical to consider while testing, automated testing is a clear solution. Further, apps must be tested not only on public releases of iOS but also on beta releases to leverage new features, not rely on any deprecated features, and detect issues early in development.

Currently, the most popular and out-of-the-box solution provided by Apple is to leverage the built-in UI Testing bundle target in the official Xcode IDE. This allows developers to compile the app in UI testing mode and enables them to develop UI tests for automated testing. These tests can be created manually by interfacing with the XCUITest API or by using the provided record tool, which allows developers to perform user interactions on an iOS device to generate corresponding XCUITest code for repeating those actions automatically. Since this recording tool writes declarative code that represents user actions, we can observe how it uniquely identifies elements of interest by traversing UI hierarchy in XCUITest's API language and use of text values/accessibility labels. As this testing target runs in isolation outside the main target and without interfacing with UIKit, it does not have access to the UIView and UIControl API methods available while developing the interface. Moreover, since the target runs outside the app sandbox, the XCUITest API does not have information about the internal state of the app either except what is presented on the screen (alongside its accessibility values). Such

automatically generated codes are therefore fragile and often need developer modifications to be accurate and reliable for all different devices and configurations that the developer intends to support. As apparent, this approach has various challenges:

(a) Testers must have access to the iOS development setup, which at-minimum includes a Macintosh computer and often access to multiple devices in configurations that they intend to support.

(b) Testers must be familiar with either Swift or Objective-C, the two languages supported for native iOS app development.

(c) Testers must also be familiar with the XCUITest library and often need to have an understanding of the underlying code architecture to create robust tests.

(d) Since automatic code generation doesn't support test assertions through GUI, along with various other limitations in the scope of what can be recorded, these actions must be written as code.

This presents a challenge to large corporations and indie developers alike as testers must be technically familiar with iOS app development to be able to generate test cases or must individually repeat their actions on all supported devices and their configurations to ensure that their app works reliably. If they choose the later manual route, then it must be repeated at least every time a new update is intended to be pushed on the App Store, which is typically once every two weeks as a rule of thumb. If errors are found, and the code is modified, then all steps must be repeated. Furthermore, good coding practices suggest continuous integration (CI) using the test case codes that automatically repeat actions to be tested across all supported devices, which is expensive and prohibitive, as discussed above. Because of the need for technical how-to, it

becomes a choice between dedicating technical talent towards new features and bug reports as experienced by users or having comprehensive test coverage. Given the real-world limitations of time and resources, the former is often picked, which is acceptable for short-term goals but exponentially increases future costs of product maintenance.

Various possible alternative solutions to the native Xcode development issue have been researched and adapted for different parts of the testing process. Appium [9] allows programmatically capturing all XCUI API information present on the screen and providing user interactions to the iOS device. WebDriver agent [10], which is now officially deprecated, and the newer IDB (iOS development bridge) [11] tool by Facebook provide developers an alternative way to execute user interactions without having to interact with Xcode GUI. These tools rely on the Xcode command line and thus require the Xcode development setup i.e. a Macintosh computer with the latest macOS and Xcode with command lines installed. Moreover, since they execute outside the app sandbox, they are limited in their scope of accessing and analyzing the internal app state and require programming knowledge along with access to a Macintosh computer and thus fail to remedy the challenges mentioned above.

1.1 Motivating Example

To motivate the tool design and its workings, consider the example of a typical calculator application (Figure 1a) that performs four standard (DMAS) operations on integers. The UI consists of a series of buttons corresponding to the 10 digits, four operands and "AC", “C”, “.”, and “=” special operations with the accessible view hierarchy in Figure 1b.

To manually check that this application performs basic operations correctly, one can imagine thinking about 2+9 must be equal to 11 (“the test case”) and then manually pressing the

button “2”, the button “+”, the button “9” and the button “=” to find that the label displays 11. There are various such basic operations that must be verified to ensure the integrity of the application from correctly performing DMAS arithmetic operations to verifying that “C” clears the label, “AC” clears the memory and that the text field displays the right value for each button press. All of these elements must be visible and all these actions must perform as expected on the various devices with varying screen sizes that the application developer intends to support. One can see the need for automated testing for obvious reasons.

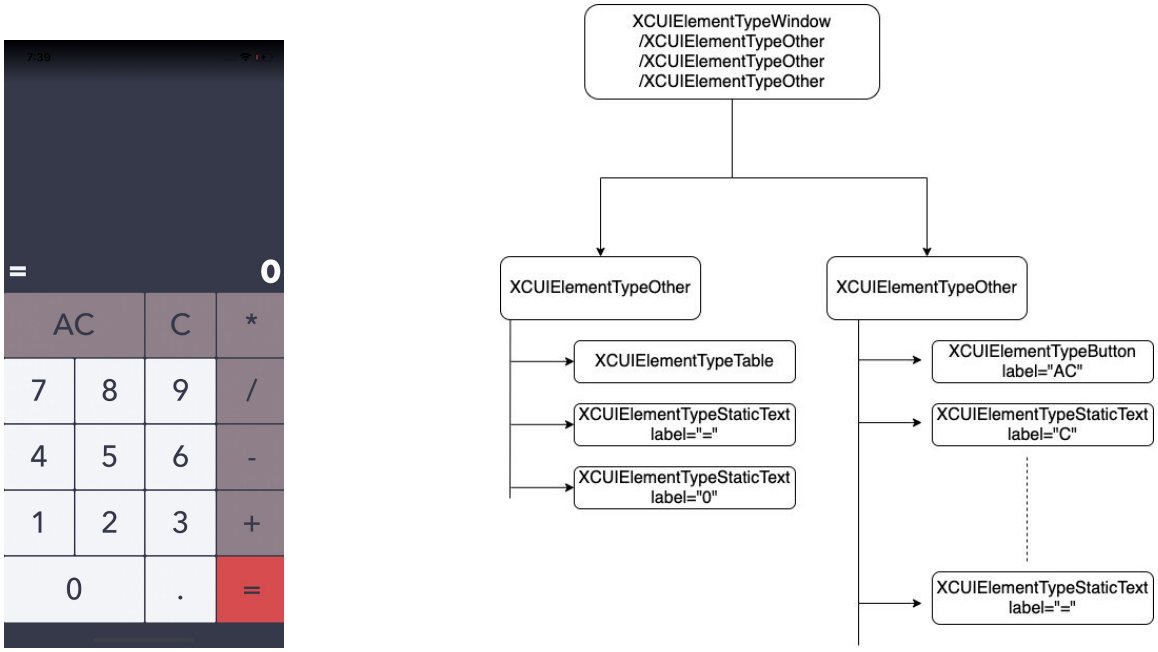

Figure 1a (left): Calculator Test Application UI and 1b (right): UI Hierarchy

1.2 Approach

To address these concerns, we present iOSTestSDK for iOS that adopts the platform-independent testing technique, as described in [4] for android apps. Specifically, we demonstrate the following features sets, as described in Chapter 3.

1. Allowing users to interact with an app on real devices and recording performed actions with expected changes in an intuitive manner.
2. Automatically encoding the recorded actions along with expected changes in a general device-independent test script.
3. Running the generated test scripts on any device configuration, both real and simulator.

The presented tool is generic and can fit different application frameworks. It can also be adapted and run in conjunction with any continuous integration pipeline used by iOS developers today. Since our tool generates XCUITest files in swift, they can be imported into the UI Testing bundle of a native app or even a dummy project as it accesses the app pre-installed on a test device based on its bundle ID and does not need access to its Xcode target. The XCUITest swift files can be replayed using the Xcode Server, adapted for WebDriverAgent applications or any of the 3rd party alternatives described in the preceding section to execute simulated user interactions programmatically.

There are several more advantages to our implementation of the platform-independent testing approach compared to the current state-of-the-art.

First, we implement the Write once, run anywhere (WORA) principle, which allows testers to record using any device and run those tests on any platform configuration. In comparison, the official solution, Xcode's UI Tester [12], uses two core technologies: the XCTest framework and the Accessibility toolkit. These technologies depend upon a lot of platform-dependent services and APIs. It requires programming knowledge with Swift or Objective-C (the two supported languages for iOS app development) and usage knowledge about these services, which is hard and time consuming based on the closed nature of iOS. Since it also

requires configuring the Xcode IDE and using a Simulator device, it highly restricts access to iOS developers with limited resources.

Second, we offer a graphical user interface (GUI) based intuitive interaction within the application available through the iOS device. These advantages allow for the tests to be generated with minimal knowledge or training and do not require any special setup or skill set. Further, the created solutions must be rapidly changed with every update to iOS, Swift, or Xcode versions, which makes it hard to keep up with the latest API versions without breaking the testing environment. The presented solution offers a fundamental way of testing, which takes care of the issues mentioned above.

Third, since we would encode the tests in a platform-independent form, the test cases are likely to be robust and would not easily break with changes that don't affect user interaction (UI) elements. In comparison, the officially supported tool, Xcode UI Tester, frequently requires developers to manually edit the auto-generated testing code for even slightly complex test cases and is bound to break with changing versions of iOS and programming languages and even with small changes within the app.

To evaluate the usability of this technique, we use a set of 5 representative open-source apps that vary in complexity from easy to typical usage to complete coverage of the available UI elements. The apps are built using Swift or Objective-C programming languages or a hybrid of both.

Calculator [13] utility would allow testing the framework against expected vs. actual results, to build commonly used assertion checks in the test script. This app is used as an example *hello-world* project for testing mobile apps

UIKit Catalog [14] is the official sample app by Apple to demonstrate different capabilities of UIKit, the supported UI framework for building GUI interactions. This app allows us to test our framework on the widely used UI Elements, representing a majority of UI Controls the framework would encounter when deployed on user applications.

XKCD [15] is an open-source Objective-C app that primarily shows xkcd comics along with their explanations and ways to bookmark them. It is primarily a UITableView based application that displays dynamically obtained data, and each cell displays detailed information about the related item. Thus, it allows us to demonstrate the testing framework's ability in a photos rich app.

You're Cancelled. [16] is a proprietary social media app available on the app store. Since it is challenging to obtain decently complex open-source social media apps that are reproducible along with the web component and that work with the latest release of Xcode., we opt to demonstrate the testing framework's capabilities on an app built by the author and made available alongside the source code for this tool. This app allows us to show how the testing framework might be used and how it performs on a primarily social network application.

iOS alarm [17] is an open-source swift application that is a clone of the official iOS alarm clock app. It is chosen for its popularity as arguably one of the most used apps on the iPhone and demonstrates the testing framework's capabilities on a general-purpose utility app.

These apps represent a wide variety of applications from a testing perspective as we evaluate common UI Controls, utility apps for assertion checks, and multi-language game apps for evaluating language-independent implementations, a popular genre for iOS apps.

The aim of this study is to show that the presented iOSTestSDK can (1) faithfully record and encode user-defined test cases (2) can encode test cases that run on multiple platforms, and (3) provides better support for automatic test generation than Xcode's UI Testing bundle. More abstractly, our goal is to considerably improve the way tests for iOS apps are generated and run, which in turn would result in an overall improvement of the iOS quality assurance process.

CHAPTER II
LITERATURE REVIEW

Smartphone apps are used for many of our daily activities including communication, businesses, utility, entertainment, gaming and other emerging domains. Similar to traditional software applications, mobile apps must be tested to ensure they behave and function as desired in a variety of environments and use-case conditions. With over 2.2 million apps available on the App Store and a projected 197 billion downloads for 2017 [18], comprehensively testing iOS apps is particularly important as it influences the lives of billions of people around the world. These tests must run on an increasing multitude of devices as Apple launches new iOS devices every year. However, as Fazzini et al discuss, mobile app testing is much human-intensive, tedious and error-prone activity and considered only by large corporations [4]. Testing iOS apps is much more complex as discussed before and indie developers, that constitute a majority of App Store sales, often overlook mobile app testing.

Popular testing techniques like manual checking and monkey testing are all naive and limited by their very style which requires little time for configuration but also results in non-reliable results. Therefore, these testing methods are just a function of probability and cannot be used to reliably test and confirm whether an app functions as desired [19].

Current research and more advanced techniques like reverse engineering, dynamic analysis, concolic testing and visual scripting are not accessible and feasible for most developers due to the lack of familiarity of such tools by app developers who do not specialize in quality assurance or software testing and the time constraints put forth due to tight project timelines. Since developers often compete between budgeting their time for testing and building new

features, often priorities are put towards business-critical new features rather than test-driven development for user interfaces. Our aim is to ease their pain and make developers and testers more productive and consume less of their time. Besides, these research tools have not been completely adopted to iOS technology and being a closed platform, these approaches are time-consuming and capital expensive as they are harder to implement and require hiring consultants to apply these tools to a development project. This forces most developers to simply overlook the testing phase in the time deficient world of rapid app development and capital constraints.

2.1 Developing iOS Apps

Developing iOS applications requires a Mac computer (macOS 10.15 or later) running the latest version of Xcode [20]. Xcode includes all the features you need to design, develop, and debug an app. Xcode also contains the iOS SDK, which extends Xcode to include the tools, compilers, and frameworks you need specifically for iOS development.

Objective-C is the primary programming language used to write software for OS X and iOS [21]. It's a superset of the C programming language and provides object-oriented capabilities and a dynamic runtime. Objective-C inherits the syntax, primitive types, and flow control statements of C and adds syntax for defining classes and methods. It also adds language-level support for object graph management and object literals while providing dynamic typing and binding, deferring many responsibilities until runtime.

Apple introduced Swift at Apple's 2014 Worldwide Developers Conference (WWDC) [22] , a successor to both the C and Objective-C languages. It includes low-level primitives such

as types, flow control, and operators and also provides object-oriented features such as classes, protocols, and generics, giving Cocoa and Cocoa Touch developers the performance and power they demand. Swift has been a focus of iOS app development since its launch and an increasing number of app developers are shifting towards the powerful programming language. Swift made it to the top 10 in the monthly TIOBE Index ranking of popular programming languages in March 2017 [23]. Although it's a powerful modern language and made specifically for developing on Apple platforms including iOS, it's relatively young and each major upgrade leads to breaking changes during compile-time and forces developers to rebuild and revise their applications every year. In comparison, Objective-C is a much stable and standard programming that has been around for about 33 years with a massive knowledge base and developer familiarity given its similarity to other C based programming languages.

Xcode supports developing apps using both Swift and Objective-C. It also provides for an interoperability layer to facilitate communication between the two languages. The entry point for iOS applications is the main function similar to every C-based program. But Xcode automatically creates this function as part of your basic project that allows for automatically importing and linking added frameworks as well as managing the app life cycle through UIApplication delegate. Figure 2 shows an example of this function. The main function passes off the operation to UIApplication which manages the App Lifecycle as illustrated in Figure 3. UIApplication calls the UIApplicationDelegate which is responsible for configuring runtime attributes for an app and launches the desired View Controller. View Controllers are Cocoa Classes, which are class or object based on the Objective-C runtime and inherit from the root

class NSObject [24], that are used to manage the presented screens and UIKit Elements including Views and UI Objects.

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"

int main(int argc, char * argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
    }
}
```

Figure 2: iOS main function [25]

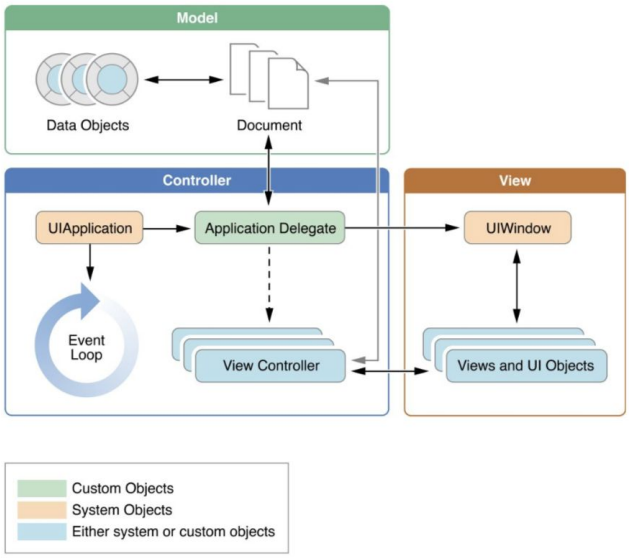

Figure 3: iOS App Architecture [25]

2.2 Frameworks

A framework is a hierarchical directory that encapsulates shared resources such as a dynamic shared library, nib files (a special type of resource file used to store the user interfaces of iOS and Mac apps [26]), image files, localized strings, header files, and reference

documentation in a single package. Frameworks serve the same purpose as static and dynamic shared libraries as they provide a library of routines that can be called by an application to perform a specific task. For example, the Foundation framework provides the programmatic interface for the Cocoa classes and methods. They are useful because:

- It allows app developers to reuse libraries of code with just a few steps of installation.
- It groups related resources and makes it easier to install, uninstall or locate these resources
- It allows for rapid exchange as frameworks reside inside an application's sandbox.
- Only one copy of a framework's read-only resources resides in-memory at any given time, regardless of how many processes are using those resources. This sharing of resources reduces the memory footprint of the system and helps improve performance.

Apple does not provide any inbuilt solution to install hybrid 3rd party frameworks in an iOS application. Therefore, the two most popular dependency manager solutions used by app developers to interact with third-party open-source libraries are CocoaPods and Carthage.

2.3 Testing Methods

2.3.1 Reverse Engineering

Reverse engineering is one of the most common methods of extracting useful information from traditional software applications. Joorabchi et al discuss a reverse engineering technique to automatically crawl through an iOS app and infer a model of its user interface states dynamically [27]. It is capable of automatically detecting unique states to generate a correct model of the given mobile application within a reasonable amount of time. However, it fails to recognize the

relevance of sequential actions for plotting an interface model and only works on a limited number of User Interaction (UI) elements and user interactions. Science direct manipulation has now become a de-facto mode for app interactions, this technique is much more suboptimal to implement given its potential benefits.

2.3.2 Dynamic Analysis

iOS applications are unique as they run on Apple's proprietary software and hardware devices. This means we have very little access to the Operation System (OS) or the hardware status. Szydlowski et al discuss the challenge of privacy issues with iOS apps and provide a prototype implementation of dynamic analysis by taking advantage of the GNU Debugger (GDB) Project Debugger to generate method call traces for iOS applications and stimulate user interface interactions [28]. Although this technique dramatically enhances code coverage compared to other techniques, the implementation is limited in its detection of popular user interaction gestures like panning and dragging and is also unstable for custom UI elements. Since the approach is only valid for Objective-C based apps, the prototype is decreasingly relevant today; especially with the rise of Swift's popularity as the choice of native language for building iOS apps.

2.3.3 Concolic Testing

Anand et al presented a fully automatic and general algorithm; and a system for generating input events to exercise apps based on concolic testing [29]. They described ACTEve (Automated Concolic Testing of Event-driven programs) to solve the branch-coverage problem through subsumption conditions between event sequences. The prototype, developed for Android

apps, demonstrated significant (64%–95%) saving in running time than the naive concolic execution technique. The proposed solution is time-saving in general as compared to monkey testing or sequentially recording app states for all possible user interface models and can be successfully implemented for basic iOS apps but it lacks generalization for the increasingly complex apps and it does not handle complex UI interactions beyond tap events. Anand et al acknowledged that there is no substantial proof of the superiority of ACTEve over AllSeqs which means a non-optimal and non guaranteed solution.

2.3.4 Visual Scripting

With dramatic advances in vision recognition lately, a universal GUI analysis can be used to generate test cases by analyzing the flow of an app. Yeh et al presented a visual approach to search and automation, Sikuli, which allows users to take a screenshot of a user interaction element and search a documentation database by visual recognition [30]. Since it operates based solely on screenshots, it can only decipher the visible screen space and is not applicable to track invisible GUI elements and UI interactions like gestures, Input/Output and state sequence. However, with today's advanced technology, the proposed prototype can be used as a secondary aid to build more comprehensive universal test scripts based on visible GUI elements linked with their properties and user interaction events performed on the specified elements.

CHAPTER III
TOOL DESIGN

3.1 Technique & Implementation

To address the issues surrounding app testing as discussed earlier and to enhance test coverage, I present iOSTestSDK that adopts the Barista approach [4] to allow developers to build device-independent test scripts by simply importing the framework requiring no other lines of code. This is a Swift & Objective-C hybrid framework with three major components similar to the Barista approach (a) the recording phase that captures user-generated events and system responses by swizzing events on UIView and UIControl classes for faithful replay and allows recording assertions on their properties (b) the encoding phase that allows sending the said recorded log to a separately hosted web server that can encode these records as a device-independent XCUITest class which can be run on any iOS device and (c) the execution phase which is simply importing the automatically generated XCUITest files into the target project's UI Testing bundle or through a dummy app that can be run on target device that has the app under test already installed. Figure 4 shows a high-level overview of the adopted Barista approach and Figure 5 describes one possible flow of test generation based on the presented tool.

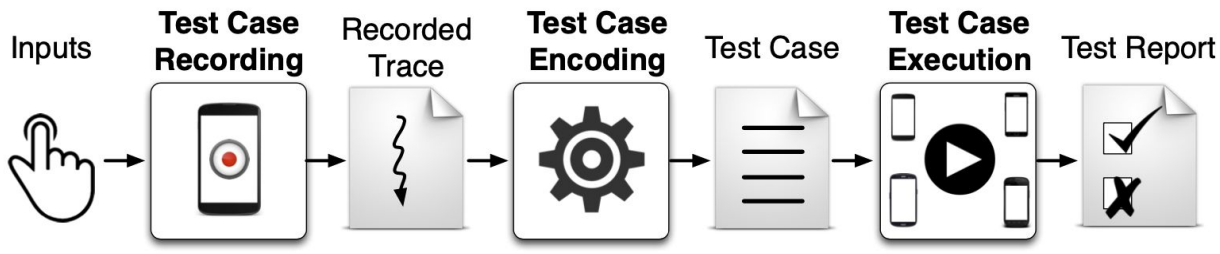

Figure 4: High-level overview of the adopted Barista approach [4]

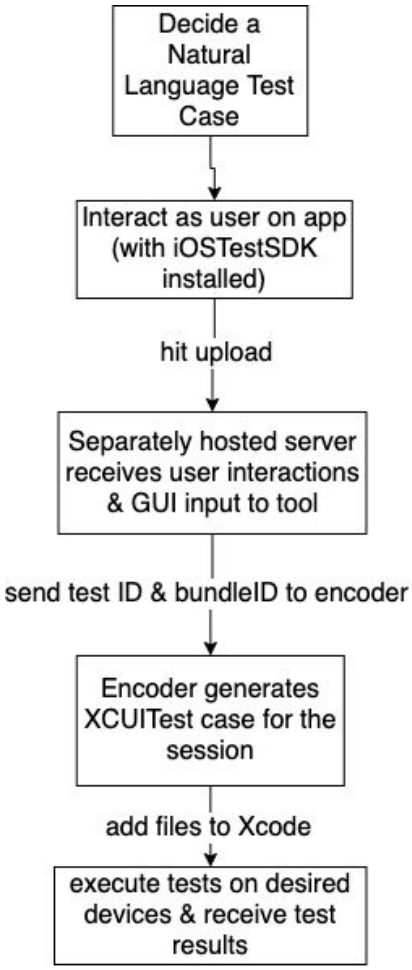

Figure 5: SDK Use Flow

Source code of the proposed framework along with the web component and reproducible results described in the next chapter are made available at https://github.com/anushkmittal/iOSTestSDK.

There are several advantages to my implementation of the Barista approach compared to the state-of-the-art:

First, iOSTestSDK implements the record once run anywhere principle that allows testers to record tests on one device and run on any platform combination instead of focusing on GUI test automation that generates tests valid only for a particular platform.

Next, iOSTestSDK supports building test cases from intuitive interactions on a real device, allowing tests to be generated with very limited knowledge and training and without any special setup or skill set. With adequate modifications, all user actions can be recorded and subsequently reproduced under similar or any arbitrary conditions.

Finally, the recorded actions and assertions can be uploaded from the device using the provided web server component and the recorded logs can then be encoded as tests in a platform-independent form without requiring Xcode or the iOS simulator. These automatically generated swift test cases can then be plugged into any existing automation frameworks or imported directly into the project target for more hands-on testing.

3.2 Recording Phase

During the recording phase, the tester adds iOSTestSDK to the application-under-tester and interacts as a user. For the “2+9=11” test case introduced in the motivating example earlier in section 1.1, the tester would press each of the 4 buttons in order and assert the text value of 11 in the results field. iOSTestSDK uses method swizzling (section 3.2.1) to capture user interactions and changes to the state of the app. This allows generating a record trace detailing each user-initiated event and the corresponding state of the app (section 3.2.2) along with any interactions with the framework toolbox such as assertion checks, screenshots or tester added comments (section 3.2.3)

3.2.1 Method Swizzling

Since iOS applications are run in their individual sandbox, it is not possible to access the internal state of application-under-test or overlay test recording toolbox by merely using a separate utility application as used in the barista approach for Android [4]. Rather, we require the framework to be imported inside the target application which allows it complete access to capture user information and action data. With access to application during runtime, we are able to leverage method swizzling, a process of changing the implementation of an existing Objective-C selector at runtime by changing how selectors are mapped to underlying functions in a class's dispatch table.

Since method swizzling can be used to inject a behavior into the view controller lifecycle, responder events, view drawing etc, we swizzle the load method that is invoked whenever a class or category is added to the Objective-C runtime. This allows us to further swizzle the action-event methods triggered on each user-initiated interactions. Swizzled action-event methods, namely `sendAction:to:from:forEvent:` and `sendEvent:`, enable the framework to capture user-initiated events before they are passed on to the application for processing and record the characteristics of user events and the state of interacted UIView(s) and UIControl (if present) before and after the user action is executed.

`sendEvent:` method intercepts incoming events to responder objects in the app. This method receives a UIEvent object that describes a single user interaction with the app. Swizzling this method allows capturing many different types of user interactions like touch, motion, remote-control events, and press events. This allows intercepting the touches (that is, the fingers on the screen) that have some relation to the event. A touch event can also include drag, multiple

touches, force touch, pan and gesture. Although we have not implemented the support for encoding these different touch events beyond a singular tap, such implementation should be trivial for specific use cases and remains a future work for the proposed SDK. In addition, this method also contains information about its superview and its bounds which allows us to detect the specific UI Element that responds to the user interaction.

`sendAction:to:from:forEvent:` is used to capture action messages identified by selector to a specified target. This allows capturing the type of UIElement that a user interacted with, it's current state like highlighted, enabled, selected and changes made by user interaction. It also allows us to capture an element's location, and it's bounds area which can be used to identify a UI Element by its location and frame to perform assertions or user events on it.

Once intercepted, key-value pairs are added to the record trace as described in the following section and action-event methods are passed on for the target application to handle. However, during the assertion mode available through the app toolbox, no user-initiated event is passed on to the target application. User interactions are used to highlight UIKit widgets of interest and capture their original properties without triggering a user-initiated event or action.

3.2.2 Generating Record Trace

The record trace is initialized with bundle ID to uniquely identify the application-under-test. On app launch, it records the unix timestamp to mark the session and system time since bootup in seconds to identify time between user interaction events for reproducibility. On each user-initiated interaction event, `sendEvent:` method intercepts the user event which allows recording the UIEventType, UITouchType, UITouchPhase along with the viewDetails at the location of touch along with its receiverSubview, receiverWindow,

superView, bounds, accessibility label, and tag. A screenshot of the application is also stored for visual verification of the application state before the event is executed if needed for later review. Additionally, the `sendAction:to:from:forEvent:` method is intercepted if the user interacted with a UIControl subclass of UIView. This allows capturing additional information about these UIKit widgets such as the isEnabled, isHighlighted, isSelected properties along with the UIControlState. Figure 6 describes a high-level overview for generating the record trace.

In the current implementation, the framework only logs interactions for UIEventTypeTouches and when the UITouchPhase is of type ended. This allows ignoring additional noise as the Objective-C runtime triggers these methods for each individual UIEvent such as UITouchPhaseBegan and UITouchPhaseMoved which could lead to hundreds of entries due to accidental touch events by the user. Furthermore, since the framework presents a toolbox in the form of a float button menu, the swizzled method checks for it using a unique accessibility label and simply ignores it for the purpose of recording user-initiated event details. Finally, if a touch is performed on a part of the screen with no UIView presented by the target application, it is simply ignored for the purpose of recording. Once these checks are performed, the framework verifies whether the application is under assertion mode which can be triggered by navigation to the assert mode using the framework toolbox on-screen and overlaying a grid. Under the assertion mode, a single tag on any UIView can be used to highlight it with a second tap to present an on-screen menu to assert any of its properties and exit the assertion mode. During the assertion mode, all user interactions are ignored and a new action log entry is created to record the assert items selected by the tester.

A new action log is initiated whenever the sendEvent: is triggered and thus, each interaction is stored within a separate action log. All action logs are stored in the “log_data” key inside the recorded trace such that they can be individually iterated and written into swift code for reproduction by the encoder. Figure 7 describes the abstract syntax of recorded trace. Figure 8 represents an example recorded trace for the “2+9=11” test case where “com.example.calculator” is the bundle ID of the application & “1590408216” is the unique test ID.

Figure 6: Flowchart representing a high-level overview of recording phase

appID	:=	*string*
manual_screenshots	:=	**(***int: string,***)***
time_since_bootup	:=	*int*
timestamp	:=	*int*
log_data	:=	**{ (accessibility_label)? (action)? (assert_items)? (bounds)? (comments)? (time_since_bootup)? (isEnabled)? (isFloatButton)? (isHighlighted)? (isSelected)? (receiverSubview) (receiverWindow) (screenshot)? (stateUIControl)? (superView) (tags)? (typeOfTouch)? (uiTouchPhase)? (viewDetails)? }**
accessibility_label	:=	*string*
action	:=	**UIEventType**
assert_items	:=	**{ (accessibility_label**: *bool***)? (bounds**: *bool***)? (isEnabled**: *bool***)? (isHighlighted**: *bool***)? (isSelected**: *bool***)? (receiverSubview**: *bool***) (receiverWindow**: *bool***) (stateUIControl**: *bool***)? (superView**: *bool***) (tags**: *bool***)? (viewDetails**: *bool***)? }**
bounds	:=	**{x y width height}**
x	:=	*int*
y	:=	*int*
width	:=	*int*
height	:=	*int*
comments	:=	*string*
isEnabled	:=	*bool*
isFloatButton	:=	*bool*
isHighlighted	:=	*bool*
isSelected	:=	*bool*
receiverSubview	:=	*string*
receiverWindow	:=	*string*
screenshot	:=	*string*
stateUIControl	:=	**UIControlState**
superView	:=	*string*
tags	:=	*int*
typeOfTouch	:=	**UITouchType**
uiTouchPhase	:=	**UITouchPhase**
viewDetails	:=	*string*
UIEventType	:=	“UIEventTypeTouches”
UIControlState	:=	“UIControlStateNormal” | “UIControlStateHighlighted” | “UIControlStateDisabled” | “UIControlStateSelected” | “UIControlStateFocused” | “UIControlStateApplication” | “UIControlStateReserved”
UITouchType	:=	“UITouchTypeDirect” | “UITouchTypeIndirect” | “UITouchTypePencil”
UITouchPhase	:=	“UITouchPhaseEnded”

Figure 7: Abstract syntax of the record trace

```
{ "appID": "com.example.calculator",
  "manual_screenshots": [__array_of_base64_strings__]
  "timesincebootup_ref": __time_since_bootup_in_seconds__
  "timestamp": __unix_time__,
  "log_data": [{ "action": "UIEventTypeTouch",
                 "isEnabled": "true",
                 "isFloatButton": "false",
                 "isHighlighted": "false",
                 "isSelected": "false",
                 "isUIControl": "true",
                 "receiverSubview": "... UILabel 'text = 2'"},
               { "action": "UIEventTypeTouch",
                 "isEnabled": "true",
                 "isFloatButton": "false",
                 "isHighlighted": "false",
                 "isSelected": "false",
                 "isUIControl": "true",
                 "receiverSubview": "... UILabel 'text = +'"},
               { "action": "UIEventTypeTouch",
                 "isEnabled": "true",
                 "isFloatButton": "false",
                 "isHighlighted": "false",
                 "isSelected": "false",
                 "isUIControl": "true",
                 "receiverSubview": "... UILabel 'text = 9'"},
               { "action": "UIEventTypeTouch",
                 "isEnabled": "true",
                 "isFloatButton": "false",
                 "isHighlighted": "false",
                 "isSelected": "false",
                 "isUIControl": "true",
                 "receiverSubview": "... UILabel 'text = ='"},
               { "isAnAssertionLog": "true",
                 "assert_items": { "View Details": "..UILabel...text = '11'.." },
           }
    ]}
```


Figure 8: Recorded trace for “2+9=11 test case”

3.2.3 Framework Usage

iOSTestSDK must be imported into the target application using a dependency manager such as Cocoapods [31] or Carthage [32], or by manually adding in the compiled framework to the target app. Once imported, a single registration call must be made in the didFinishLaunchingwithOptions method of the app delegate after UIWindow has been initialized as shown in figure 6. This allows you to provide the bundle ID of the app you wish to re-run test on (if it's different than the one being used), if screenshots should be logged after every user-initiated action, if all touch events should be recorded, if the float button with iOSTestSDK options should be visible that allows user actions such as adding comments and uploading session through GUI as described below, if the recorded actions are also recorded on console (in case of debugging) and whether the screenshots on user-initiated actions should be saved on device permanently.

```
#if DEBUG
import iOSTestSDK
#endif
@UIApplicationMain
class AppDelegate {
  func .. didFinishLaunchingWithOptions .. () {
    #if DEBUG
        iOSTestSDKManager.shared.login(applicationID: "com.example.calculator",
                                       log_screenshots: true,
                                       showOptions: true,
                                       verbose: false,
                                       saveScreenshotsLocally: true)
    #endif
    return true
 }
}
```

Figure 9: Importing iOSTestSDK in a sample open-source Calculator app

The framework also offers an intuitive graphical user interface in the form of a non-intrusive float button that can be dragged and manipulated on-screen (see figure 10a and 10b). The button can be dragged on the screen to reveal any hidden app view and testers can intuitively:

(a) Add assertion checks any UIView or UIControl present on-screen against all possible property values. For example, whether a button or a label should have a particular text, or if a control should have the exact frame, or if their subviews or super views should match etc. Ability to add assertions on all possible objective-c properties is provided out of the box. Testers can select the "Check" button to enter the add assertion mode (figure 11a) and select any UIView or UIControl with eliciting user response (figure 11b) to see it highlighted and finally another tap on the same button opens up the add assertion menu (figure 11c) where any of the desired property can be selected and user can save the selection by tapping the outside screen to dismiss. Table 1 describes a list of assertable properties derived from the UIView property values

(b) Add comments to add meaning to user-initiated actions or assertion checks (Figure 12).

(c) Take manual screenshots at any point through the test session

(d) Upload the test session and start a new one

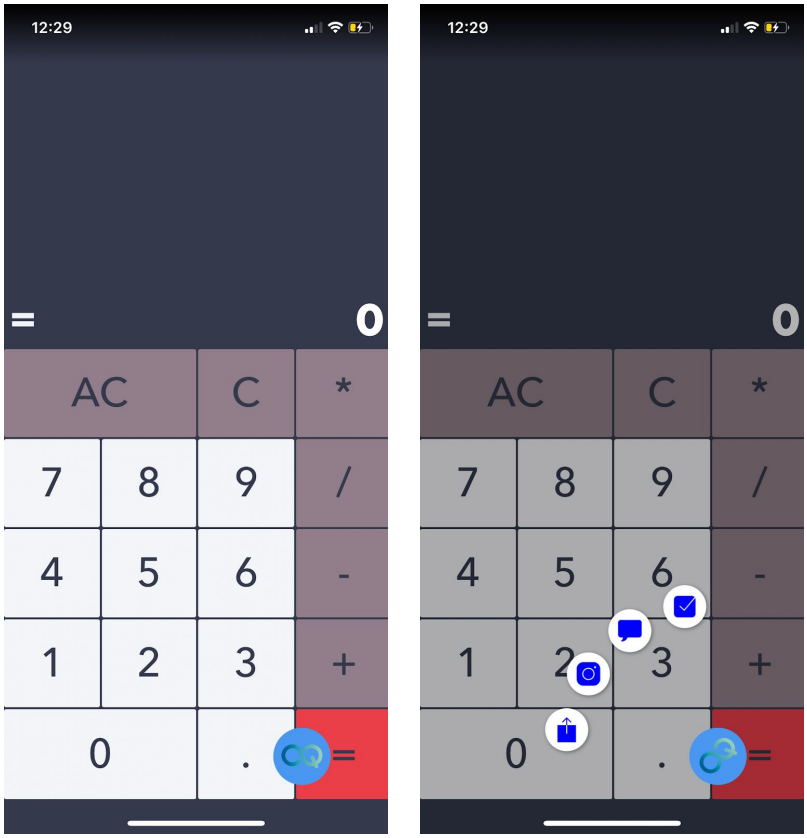

Figure 10(a) iOSTestSDK float button in calculator open-source app and Figure 10(b):

Float button presented with options

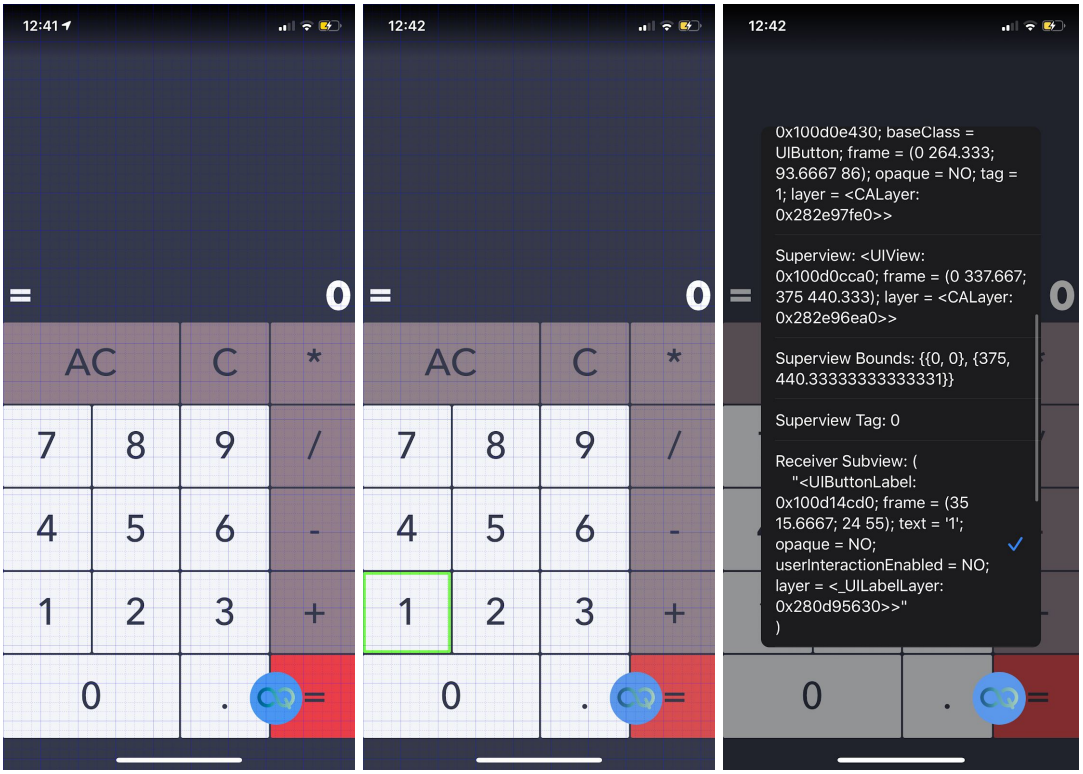

Figure 11(a) assertion check grid presented after selecting the assertion check button; 11(b) Button “1” selected in the assertion mode, single tap; 11(c) Double tapping presents an assertion table, where the receiver subview has been selected to assert property. [from left to right]

Property	Description
Enabled	Whether the UIControl is enabled
State of the Element	UIControlState constants describing the state of a control [33].
Selected	Whether the UIControl is selected
Highlighted	Whether the UIControl is highlighted
Bounds	The x,y,width,height bounds of the UIView
Tags	Optional integer tag integer associated with the UIControl
Accessibility Label	Optional label associated with the UIControl
View Details	View description including the text value of the UIView
Superview	Parent view description including the type of UIView and its text value
Receiver Subview	Child view description including the type of UIView and its text value
Receiver Window	Child window description including any views contained within and their types & values.

Table 1: Assertable properties for the test case recording phase

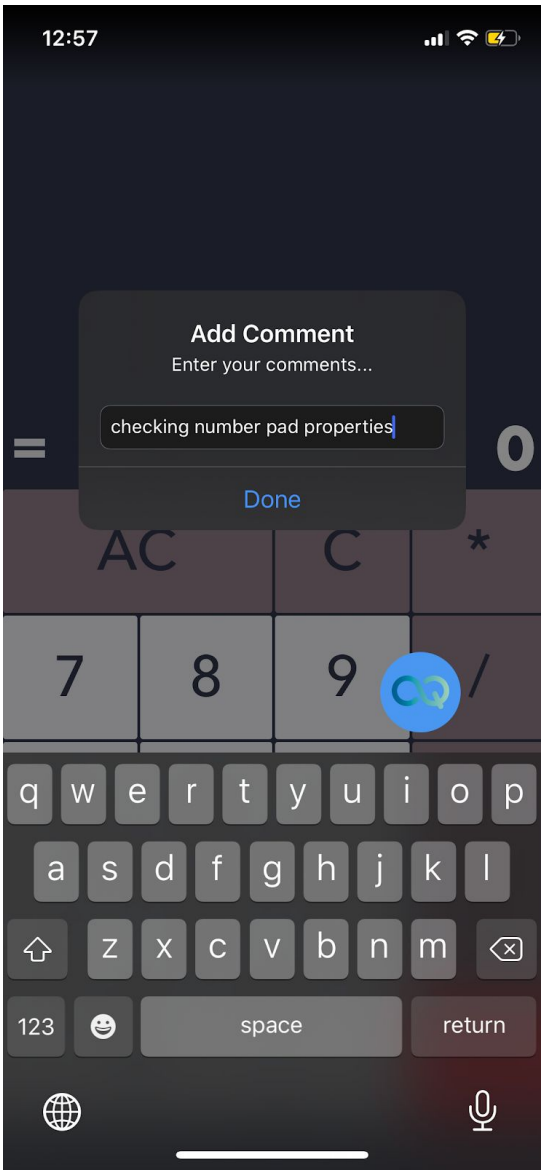

Figure 12: Add comments screen presented with user typed in comment

3.3 Encoding Phase

Once the recording phase ends, the record trace can be made available for encoding. We present a Flask server built in python that can be run alongside ngrok to expose localhost over the internet and allows uploading test sessions from a real device along with iOS simulators. The python based encoder service can then be run with specified `appID` and `sessionID` of the desired app and it would produce a XCTestCase swift file based on the recorded session by parsing each user-initiated interaction one at a time, (see Figure 14 for pseudocode):

First, it attempts to identify the type of known element by first looking for base class property in view details matching UIButton or UILabel, then the existence of UILabel, UIButton, UIButtonBarButton, UIAlertControllerActionView, or UITableViewLabel in view details, then TableViewCell in superview and finally UIBackButtonContainerView in receiver subview. These manual checks have to be performed as XCUITests are performed outside the application sandbox and do not have access to UIKit elements. Apple provides XCUIElementTypes for all possible UIKit elements which are accessible during UI testing. This approach provides support for XCUIElementTypeButton, XCUIElementTypeStaticText, and XCUIElementOther cells which covers most of the basic app requirements as demonstrated in the next chapter on experiments but future work remains to provide a complete class coverage between UIKit and XCUIElement detection.

Next, with the XCUIElement type known, the parser attempts to locate the desired element based on known property values. Unlike the Barista approach in [4]., Xcode XCUITesting doesn't provide native support for XPath querying and so elements must be individually referred to and retrieved using accessibility values or label values associated with

them. In that spirit, the parser first looks for any accessibility label provided by the developer to the element of interest. However, as [4] establishes, most developers typically do not provide these values for all elements in their apps, we look forward to the receiver subview and view details to identify any text or title value associated with the element. With the element type and the value known, a reference is generated to the desired element or an error is produced being unable to query the element due to inadequate information or missing support.

Finally, based on whether the log is of type assertion (i.e. it is simply a check against the user-specified properties of the element) or user-initiated event (i.e. a tap action occurred)., the parser diverges in two different directions. For a user-initiated event, it simply writes the code to verify whether the element exists on screen and then taps it to mimic the user action. For an assertion event, the parser goes through every assertion item checked by the user and extracts and compares the expected value to the value of that property for the identified element. This has been implemented for the selected, enabled, Accessibility Label, State of Element, Receiver Subview and View Details properties of any given element. For the later two., only the checks against text values have been enabled as a proxy to verify whether the selected element matches the expected display value or not. There are various other properties that can be added to verify here depending on the use case and are left as ~~a~~ future work.

In selecting the elements and identifying their values, a few redundant calls had to be added to keep the code sufficiently abstract allowing myriad use cases. To support this, elements are verified against possible false positives and duplicates, which generates some extra code. These can be minimized based on rigorous detail to the state of the app and presence of elements

on-screen but cannot be done without access to Xcode during the encoding phase and are beyond the scope of the proposed tool.

Once all recorded logs have been iterated through, the encoding phase completes and a swift file is automatically generated containing the XCUITest code for testing the session. For our motivating example of the "2+9=11" test case introduced in section 1.1, the supplied recorded trace would automatically generate an XCUITest file corresponding to the user actions as shown in figure 13.

```
import XCTest
class iOSTestSDK1590408216: XCTestCase {
    func testSession() throws {
        let app = XCUIApplication(bundleIdentifier: "com.example.calculator")
        app.launch()
        // check & press the button "2"
        let el0 = app.buttons.staticTexts["2"]
        XCTAssertTrue(el0.exists)
        el0.tap()
        // check & press the button "+"
        let el1 = app.buttons.staticTexts["+"]
        XCTAssertTrue(el1.exists)
        el1.tap()
        // check & press the button "9"
        let el2 = app.buttons.staticTexts["9"]
        XCTAssertTrue(el2.exists)
        el2.tap()
        // check & press the button "="
        let el3 = app.buttons.staticTexts["="]
        XCTAssertTrue(el3.exists)
        el3.tap()
        // check that label "11" is present
        let el4s = app.staticTexts.containing(NSPredicate(format: "label LIKE[c] '11'"))
        let el4 = filterButtonLabels(el4s)
        XCTAssert(correct_el4.count == 1)
        XCTAssert(correct_el4[0].exists)
        app.terminate()
    }
}
```

Figure 13: Automatically generated XCTestCase for the "2+9=11 test case"

```
def generateXCTestCase(recorded_trace):
    // start a new empty file
    output_file = []
    // iterate through each user initiated event
    for each user_event, idx in recorded_trace:
        // find the matching XCUIElement to UIKit widget
        element_type = findXCUIElement(user_event)
        // find the best identifier to uniquely identify the XCUIElement
        element_identifier = getIdentifier(element_type, user_event)
        // write swift code for identifying element
        switch element_type:
            if element_type == XCUIElementTypeButton:
                output_file -> "let el{{idx}} = app.buttons['{{element_identifier}}']"
            if element_type == XCUIElementTypeStaticText:
                output_file -> "let el{{idx}} = app.staticTexts['{{element_identifier}}']"
            if element_type == XCUIElement:
                output_file -> "let el{{idx}}s = app.cells.containing(NSPredicate(format: "label LIKE[c]
'{{element_identifier}}''))"
          if element_type == XCUIElementTypeBackButton:
                output_file -> "let el{{idx}} = app.navigationBars.buttons.element(boundBy: 0)"
         // if this is a user action, write code to reproduce
         if user_event.TYPE == USER_ACTION:
                output_file -> “XCTAssertTrue(el{{idx}}.exists)”
                output_file -> “el{{idx}}.tap()”
         // if this is an assertion, write code to match value
         else if user_event.TYPE == ASSERTION:
                for assertion_log in user_event:
                    // store the expected value of the property from record trace
                    output_file. -> “let expected_val{{idx}} = {{assertion_log.value}}”
                    // retrieve the true value by querying property of the element
                    output_file -> “let true_val{{idx}} = element.{{assertion_log.item}}”
                    // assert that expected value matches true value
                    output_file -> “XCTAssertTrue(expected_val{{idx}} == true_val{{idx}}”

def findXCUIElement(user_event):
    if user_event on UIButton | _UIAlertControllerActionView | _UIBackButtonContainerView | _UIButtonBarButton:
        return XCUIElementTypeButton
    if user_event on UILabel:
        return XCUIElementTypeStaticText
    if user_event on TableViewCell:
        return XCUIElement
    if user_event on UIBackButtonContainerView:
        return XCUIElementTypeButton

def getIdentifier(element_type, user_event):
    if check_accessibility_label(element_type):
        return accessibility_label(element_type)
    if superview in user_event == _UIAlertControllerActionView:
        return title_value(element_type)
    if view details in user_event | receiver subview in user_event:
        return text_value(element_type)
```

Figure 14: Pseudocode for encoder service that reads record trace JSON to write XCTestCase swift file

3.4 Execution Phase

Since the generated XCTestCase is a human-readable swift file, it can be imported through any of the myriad ways in which a developer might have their existing development workflow set up. This includes (1) manually dropping the generated XCTestCase file in the "UI Testing" target of the Xcode project of application-under-test; (2) creating a blank application with just the "UI Testing" target that can be run on any device that has the application-under-test pre-installed as the generated code searches for that application on device by its bundle ID and thus if the app already exists on the device, it doesn't need to be installed again; (3) using Xcode server / CLI to clean install application-under-test on any device configuration and then running the generated XCTestCase; and any other workflows.

Although the generated tests are complete, testers or developers can further develop or enhance the automatically generated test cases by simply editing the swift file for XCTestCase of interest. Figure 13 presents steps to add generated test files in an Xcode project & Figure 5d presents a sample generated XCUITest replayable file for 2+9=11 assertion test for the calculator app, in continuation to the figures above.

Finally, the following chapter demonstrates the use of iOSTestSDK on a few sample apps demonstrating the above-detailed phases.

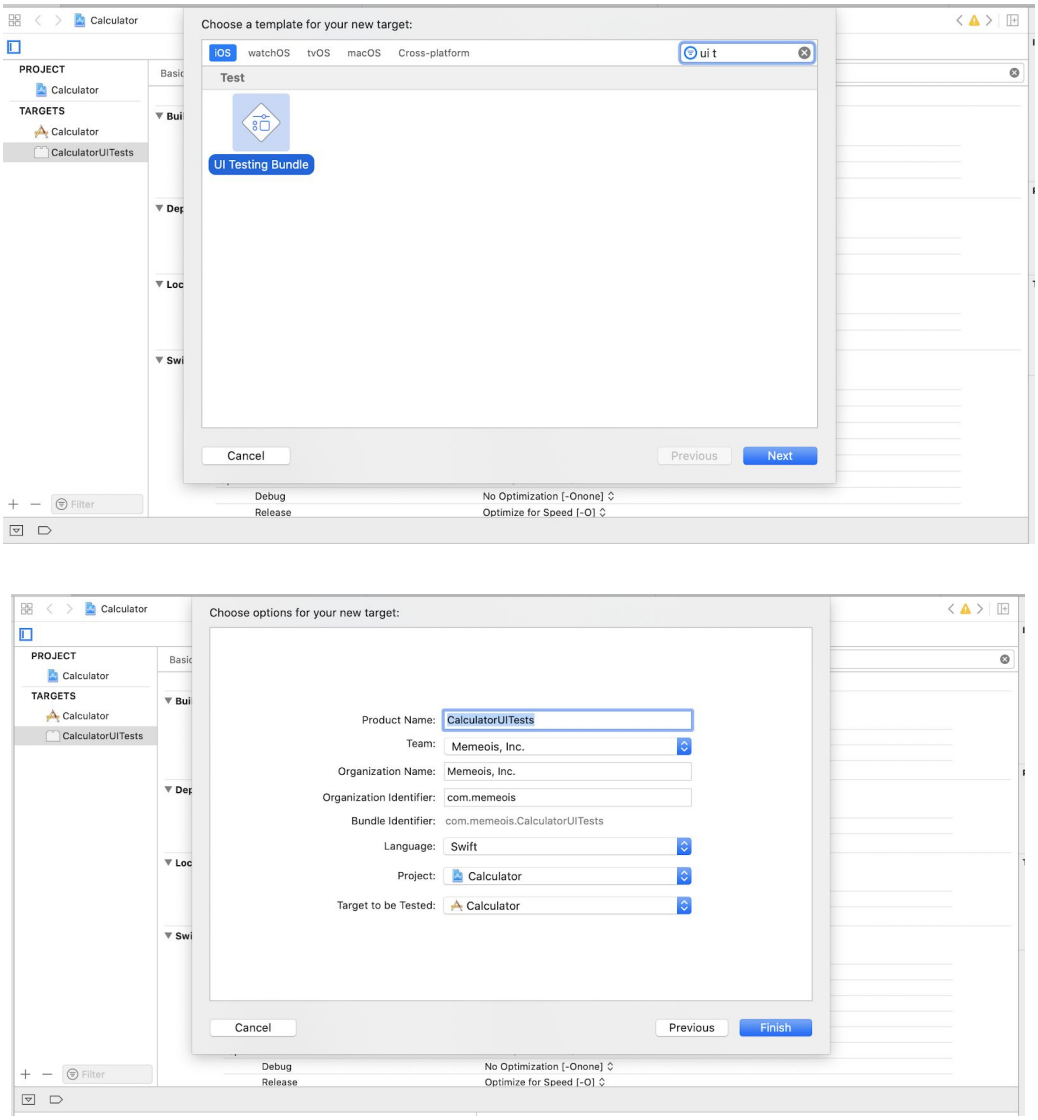

Figure 15: Adding UI Testing Bundle target to an existing Xcode project.

3.5 Similarities and differences to Barista approach

Similar to the Barista approach, the presented iOSTestSDK works in 3 phases.

First, the recording phase where a tester can naturally interact with the application-under-test as a user and can add assertions using our tool's option menu. This phase outputs a recorded trace similar to the Barista approach. However, the recorded trace of our

presented tool captures properties of UIControl & UIView which is dissimilar from the android counterpart such as absence of i-type properties, interaction-def, and most notably XPath as iOS APIs do not provide XPaths nor do they allow the use of Xpaths for querying elements. Instead, the framework records details of view, subview, and superview and their associated values to uniquely identify an element if its accessibility ID is not present.

Second, the encoding phase takes the recorded trace as input and outputs a generated test case akin to the barista approach. However, since iOS applications do not run based on activities, we do not require a two-phased set-up and steps based approach. We do utilize a similar three-phase approach for encoding interactions. (1) retrieve the element based on values of the identified XCUIElement type rather than the "selectors" (2) identifying the action (3) providing the parameters for the action. Since we only support UITouchesPhaseEnded, steps (2) and (3) are trivial but can be expanded to accommodate more complex user interactions.

Finally, the encoded test cases can then be run by importing the generated XCUITest swift files into the UI Testing target of a project. These test cases can be run in a fashion similar to the Barista approach by configuring various devices to concurrently run the generated test cases automatically. However, we do not provide any special tools to execute these test cases as it is beyond the scope of this paper.

CHAPTER IV
Experiment & Results

4.1 Experimental Setup

To analyze and test the effectiveness of our method and to assess the expressiveness, efficiency, and ultimately usefulness of the approach, we chose 5 open-source iOS applications that are recent and compatible with the current Xcode 11.5. This eliminated apps that are built using Swift versions below 4.0 or that do not compile with Xcode 11.5 on macOS Catalina (11.15.4) as these are basic requirements required by Apple for any active iOS app distributions through AppStore Connect. Moreover, we chose only those applications that are sufficiently complex i.e. have more than 10 elements in the app to rule those that do not represent typical app store apps. We excluded open-source apps that require complementary web components to install and function as they are overly complex to setup and beyond the scope of this work. With these requirements, we searched GitHub and selected 5 native iOS applications built either using Swift or Objective-C (or a hybrid of both). Once selected, for each of these apps, we create 5 natural language test cases (NLTCs) that a typical Quality Assurance engineer would want to check and ensure that they replicate on all supported devices. These are shown in Table 1 below.

Calculator [13] is a basic 10 digit 4 function calculator app that acts as a “hello world” project to demonstrate simple use cases for iOSTestSDK.

UIKit Catalog [14] is the official UIKit sample code provided by Apple that demos the capabilities of UIKit elements. This allows us to test our framework on the widely used UI

Elements, representing a majority of UI Controls the framework would encounter when deployed on user applications.

XKCD [15] is an open-source Objective-C app that primarily shows xkcd comics along with their explanations and ways to bookmark them. It is primarily a UITableView based application that displays dynamically obtained data and each cell displays subsequent information about the related item. Thus, it allows us to demonstrate the testing framework's ability in a photos rich app.

You're Cancelled. [16] is a proprietary social media app available on the app store. Since it is challenging to obtain decently complex open-source social media apps that are reproducible along with the web component and that work with the latest release of Xcode., we opt to demonstrate the testing framework's capabilities on an app built by the author and made available alongside the source code for this tool. This app allows us to show how the testing framework might be used and how it performs on a primarily social network application.

iOS alarm [17] is an open-source swift application that is a replication of the official iOS alarm clock app. It is chosen for its popularity as arguably one of the most used apps on the iPhone and demonstrates the testing framework's capabilities on a general-purpose utility app.

Serial No.	App Name	App Store Category	Primary Language	NLTCs created to test the app	Source
1	Calculator	Utility	Swift 4	1. Check that 2+9=11 2. Check that AC clears to 0 3. Check that AC clears memory 4. Check that "+" button is enabled throughout operation 5. All buttons are present and clickable by a user	https://github.com/Mackis/Calculator
2	UIKit Catalog	-	Swift 5	1. be able to navigate through any table view cell of choice and come back. 2. be able to go to a new view and click a UI element button 3. be able to pick any choice on all kinds of UI Alerts w/o encountering any error. 4. be able to interact with action sheets and verify that all choices are selectable and do not produce an error. 5. be able to verify text in a UIStepper label	https://developer.apple.com/library/content/samplecode/UICatalog/Introduction/Intro.html
3	XKCD	Photos & Videos	Objective-C	1. be able to verify that round buttons are present 2. be able to tap on any round button 3. verify the title displayed on top of app is complete and present 4. be able to tap on a round button and then check the explain button is present and is tappable. 5. verify the explain button label, the button name and it's date and the message are present when a round button is clicked.	https://github.com/mamaral/xkcd-Open-Source
4	You're Cancelled.	Social Networking	Swift 5	1. be able to tap the add button and reach add screen. 2. be able to tap save on the add screen and receive the empty fields message. 3. verify that plans are visible on-screen on load and after coming back from the add screen 4. verify that the navigation bar is visible with both bar buttons and title displaying correctly. 5. be able to go inside a plan, press cancel, see the waiting for friend label, verify the nav bar title to be the plan name, come back.	https://github.com/anushkmittal/iOSTestSDK/tree/master/OS%20test%20apps
5	iOS alarm	Utility	Swift 4	1. verify the nav bar title and two bar button are present 2. verify that the alarms set are present 3. be able to tap the add button and tap "cancel" or "save" to exit the add alarm screen. 4. be able to tap the "edit" button and "done" button displays then tap the "done" button to see "edit" button again. 5. tap add button to reach the alarm screen and verify that title and all present options and values are present.	https://github.com/natsu1211/Alarm-ios-swift

Table 2: List of open-source apps & NLTCs used for experimentation

The NLTCs created for each of these apps are then recorded, each as an individual session, by running the target app with iOSTestSDK installed on an iPhone X device with iOS 13.5. Since 92% of all devices introduced in the last four years use iOS 13 [34] and this device was readily accessible to the author, this configuration was chosen to record the tests. There should not be any material impact between different devices used for recording. The flask server was configured on localhost made public through ngrok such that the real device can upload recorded sessions which are then stored on the computer system. Once all the NLTCs were recorded for a particular app, encoder jupyter notebook was run, manually changing the `appID` and `sessionID` to match each of the 5 NLTCs for a given app. This generated 5 XCUITest swift files in the folder containing the encoder jupyter notebook that were then manually dragged into an added UI Testing target (if it didn't already exist) for the project in Xcode. Since bridging Swift code on a separate target with an Objective-C only app is unavailable in Xcode 11.5, a blank test runner app was created with added UI Testing target to add the XCUITest case files for XKCD application.

Once all of the 5 XCUITest session swift files had been added to the UI Testing target in Xcode., they were run on iPhone XS Max, iPhone X, iPhone 6s Plus, iPhone 6s, and iPhone SE along with 3rd generation iPad Pro and iPad Pro. These devices were chosen to represent the varying screen sizes, different generational devices, and most importantly all the classes of devices that app developers are required to support in order to publish their app through App Store [20]. iPhone X was chosen as the tests were initially recorded through that device and it acted as a sanity check against the testing framework's capability to reproduce user action in a similar environment. We ran the tests on iPhone X, iPhone 6s, and iPad Pro real devices as they

were physically available to the author while others were run through the Xcode simulator. All devices were configured to run on iOS 13.5, the latest public release of iOS.

4.2 Results

All of the recorded logs and generated XCUITest swift files are available to view in our github repository here: https://github.com/anushkmittal/iOSTestSDK/tree/master/Results. Primarily, we were able to replicate the tests on all devices under consideration for each of the selected test apps. To check the validity of these tests, we also manually inspected each of the generated swift files and stepped through the code to verify whether the code actually checks for the elements of interest and their properties.

Table 2 presents a list of all devices vis-à-vis test apps. Figure 12 depicts the UI widgets handled by our presented tool & the test cases aforementioned. We focus on happy paths to explore the various possible use cases of the presented SDK and invite the open-source community and further academic research into testing the limits of the presented framework. Some of the currently identified limitations include:

- inability to handle multi-touch gestures

- missing support for split-screen multi-touch apps for the newly released iPadOS

- need to expand the scope of UIControls covered to include recording user interactions on UIPicker and UIStepper.

- support for the newly introduced SwiftUI. Since SwiftUI forces refresh every time the datasource is manipulated, the GUI for iOSTestSDK does not work out of the box when adding through AppDelegate on top of the window.

Device	Screen Size	Device Type	Calculator	UIKit Catalog	XKCD	You're Cancelled.	iOS alarm
iPhone XS Max	6.5-inch	Simulator	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed
iPhone X	5.8-inch	Real	recorded on device + executed	recorded on device + executed	recorded on device + executed	recorded on device + executed	recorded on device + executed
iPhone 6s Plus	5.5-inch	Simulator	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed
iPhone 6s	4.7-inch	Real	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed
iPhone SE	4-inch	Simulator	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed
iPad Pro (3rd gen)	12.9-inch	Simulator	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed
iPad Pro	11-inch	Real	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed	all 5 test cases executed

Table 3: List of devices and recording / execution status

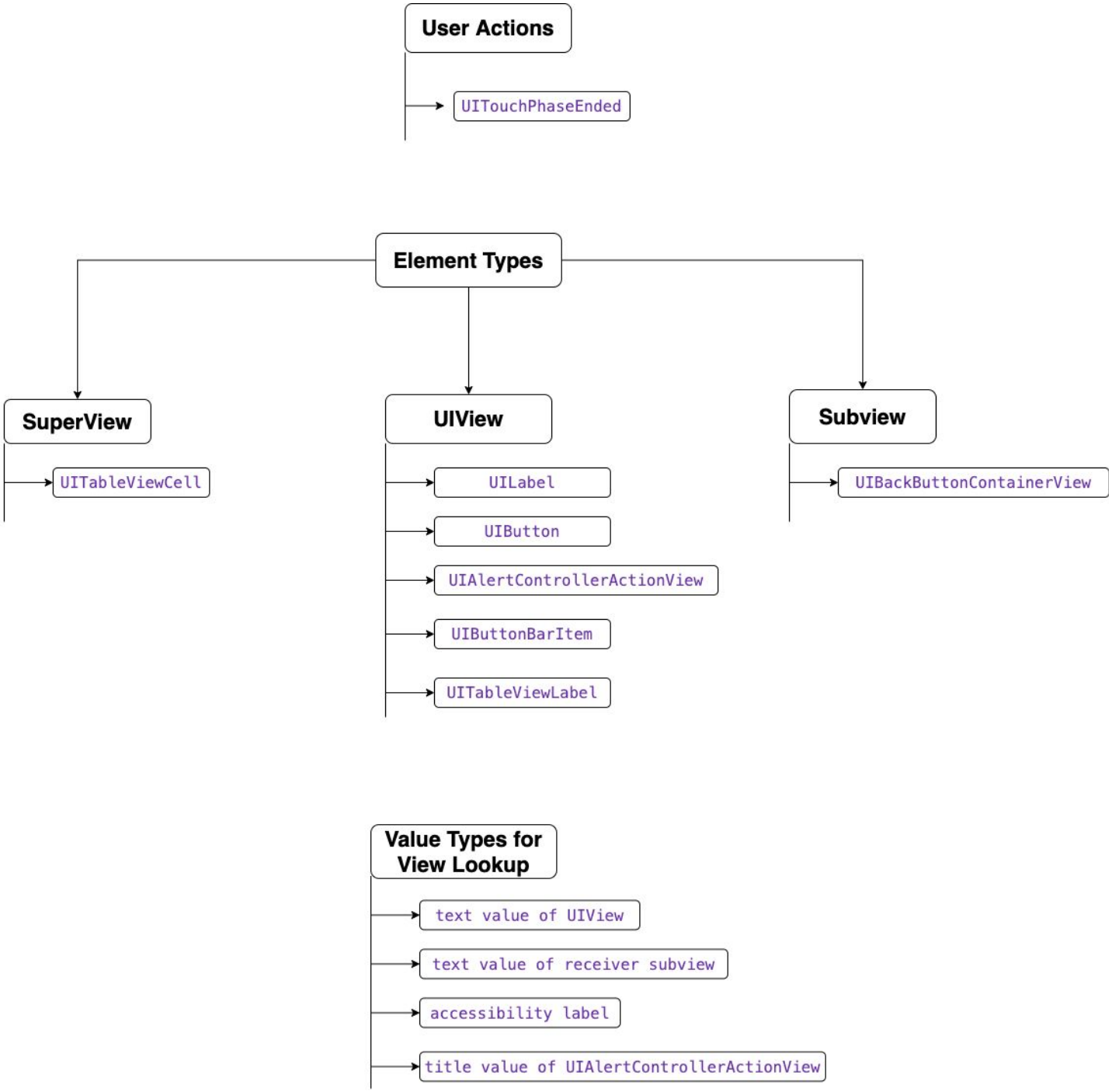

Figure 16: UI widgets handled by iOSTestSDK

CHAPTER V
CONCLUSION

We have presented a proof of concept of a device-independent testing framework that runs across all possible iOS device configurations and reliably tests across a wide range of UI elements typically used in real-world scenarios. This tool adapts the Barista approach [4] and demonstrates a GUI based technique that can leverage the underlying OS architecture through function swizzling and offer insights on user actions and app state passively. The results demonstrate potential application and effectiveness of the proposed solution, capable of automatically generating large form tests simply based on user actions and optionally any assertion checks in-between actions. Potential limitations include the limited state space covered for detecting XCUIElement types, determining the exact element based on XCUIElement hierarchy wherein each element of hierarchy must be converted to its corresponding XCUIElement, the scope of encodable assertion items (while all are recorded only the most prominent are made available in the encoder with future work remaining on other assertion items). This technique is also limited in its current form for the increasingly popular react-native cross-platform applications. Moreover, the framework has not been adapted for nor extensively tested against SwiftUI, the new development paradigm introduced in last year's WWDC. Since SwiftUI automatically refreshes the presented view based on published changes in environment objects and the UIWindow is not guaranteed to remain persistent, the testing framework's float button must be added in as a ZStack element overlaid in the content view some view structure. This limits the easy installation of testing requiring developer changes to the very structure of the app under testing.

CHAPTER VI
FUTURE WORK

We envision a number of possible directions for future work. We plan to extend the current evaluation by (1) performing a user study with a large number of experienced iOS developers on a greater variety of iOS applications and (2) running the generated test cases on a broader spectrum of the iOS platform versions and device configurations.

We will also extend our technique in several ways. First, we will add to the framework an ability to factor out repetitive action sequences, such as app initialization, and allow testers to load these sequences instead of having to repeat them for every test. Second, we will investigate how to add sandboxing capabilities, so that it can generate tests that are resilient to changes in the environment. Third, based on feedback from developers, we will extend the set of assertable properties that testers can use when defining test cases. Fourth, we will investigate the use of fuzzing for generating extra tests in the proximity of those recorded, possibly driven by specific coverage goals. Fifth, we will study ways to help developers fix broken test cases during evolution (e.g., by performing differential analysis of the app's UI). Finally, we will consider the use of our technique to help failure diagnosis; a customized version of that could be provided to users to let them generate bug reports that allow developers to reproduce an observed failure.

In June 2020, Apple announced macOS Big Sur along with Apple silicon that would allow developers to make their iOS and iPadOS apps available on the Mac without any modifications [35]. These "Universal 2" apps can make use of our presented framework and tools out-of-the-box with slight or no modifications allowing our framework to test apps across all supported apple platforms using our WORA approach. Since the program is still in beta,

much work remains in extensively testing the existing workflow but the use of low-level swizzling techniques along with automatic translation for Universal 2 apps [35] should allow developers to adopt our tool and technique for these newer use cases.

REFERENCES

[1] Clement, J. “Number of Apps in Leading App Stores.” *Statista*, 4 May 2020, www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores.

[2] Clement, J. “Mobile App Revenues 2014-2023.” *Statista*, 1 Aug. 2019, www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast.

[3] “Americans Don't Want to Unplug from Phones While on Vacation, Despite Latest Digital Detox Trend.” *Asurion*, 20 July 2019, www.asurion.com/about/press-releases/americans-dont-want-to-unplug-from-phones-while-on-vacation-despite-latest-digital-detox-trend.

[4] Fazzini, Mattia, et al. "Barista: A technique for recording, encoding, and running platform independent android tests." *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017.

[5] “Apple's App Store Ecosystem Facilitated over Half a Trillion Dollars in Commerce in 2019.” *Apple Newsroom*, 2 July 2020, www.apple.com/newsroom/2020/06/apples-app-store-ecosystem-facilitated-over-half-a-trillion-dollars-in-commerce-in-2019.

[6] GlobalStats, StatCounter. "Mobile and tablet internet usage exceeds desktop for first time worldwide." *http://gs.statcounter.com/press/mobile-and-tablet-internet-usageexceeds-desktop-for-first-time-worldwide* (2016).

[7] Clement, J. “App Stores: Number of Apps in Leading App Stores 2020.” *Statista*, 4 May 2020, www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores.

[8] O'Dea, Published by S. “Share of People with IPhone in the US 2014-2021.” *Statista*, 27 Feb. 2020, www.statista.com/statistics/236550/percentage-of-us-population-that-own-a-iphone-smartphone/ .

[9] Verma, Nishant. *Mobile Test Automation With Appium*. Packt Publishing Ltd, 2017.

[10] Facebook. “Facebookarchive/WebDriverAgent.” *GitHub*, 26 Dec. 2019, github.com/facebookarchive/WebDriverAgent.

[11] Facebook. “Facebook/Idb.” *GitHub*, github.com/facebook/idb.

[12] “Testing with Xcode.” *User Interface Testing*, 24 Jan. 2017, developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/09-ui_testing.html.

[13] Malcolmkmd. “Malcolmkmd/Calculator.” *GitHub*, github.com/Mackis/Calculator.

[14] “UIKit Catalog: Creating and Customizing Views and Controls.” *UIKit Catalog: Creating and Customizing Views and Controls | Apple Developer Documentation*, developer.apple.com/library/content/samplecode/UICatalog/Introduction/Intro.html.

[15] Mamaral. “Mamaral/Xkcd-Open-Source.” *GitHub*, github.com/mamaral/xkcd-Open-Source.

[16] Anushk Mittal. “anushkmittal/iOSTestSDK.” *GitHub*, github.com/anushkmittal/iOSTestSDK/tree/master/OS%20test%20apps/yourcancelled-master

[17] natsu1211. “natsu1211/Alarm-Ios-Swift.” *GitHub*, 21 Feb. 2019, github.com/natsu1211/Alarm-ios-swift.

[18] Poushter, Jacob. "Smartphone ownership and internet usage continues to climb in emerging economies." *Pew Research Center* 22.1 (2016): 1-44.

[18] Clement, J. “Annual Number of Mobile App Downloads Worldwide 2019.” *Statista*, 17 Jan. 2020, *www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/*.

[19] Khalid, Hammad, et al. "What do mobile app users complain about?." *IEEE software* 32.3 (2014): 70-77.

[20] *Apple Help Library*, help.apple.com/app-store-connect/#/devd274dd925.

[20] *Start Developing IOS Apps (Swift): Jump Right In*, 8 Dec. 2016, developer.apple.com/library/content/referencelibrary/GettingStarted/DevelopiOSAppsSwift/index.html#//apple_ref/doc/uid/TP40015214-CH2-SW1.

[21] *Programming with Objective-C* . 17 Sept. 2014, developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html.

[22] Apple, Inc. “WWDC 2014.” *WWDC 2014 - Videos - Apple Developer*, developer.apple.com/videos/play/wwdc2014/101.

[23] TIOBE Index for March 2017. Retrieved from https://www.tiobe.com/tiobe-index.

[24] *Apple Developer Documentation*, developer.apple.com/documentation/objectivec/nsobject.

[25] “UIKit.” *UIKit | Apple Developer Documentation*, developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html.

[26] *Apple Developer Documentation*, developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/NibFile.html.

[27] Joorabchi, Mona Erfani, and Ali Mesbah. "Reverse engineering iOS mobile applications." *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012.

[28] Szydlowski, Martin, et al. "Challenges for dynamic analysis of ios applications." *Open Problems in Network Security*. Springer, Berlin, Heidelberg, 2012. 65-77.

[29] Anand, Saswat, et al. "Automated concolic testing of smartphone apps." *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 2012.

[30] Yeh, Tom, Tsung-Hsiang Chang, and Robert C. Miller. "Sikuli: using GUI screenshots for search and automation." *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. 2009.

[31] *CocoaPods.org*, cocoapods.org.

[32] “Carthage/Carthage.” *GitHub*, github.com/Carthage/Carthage.

[33] *Apple Developer Documentation*, developer.apple.com/documentation/uikit/uicontrolstate.

[34] Apple, Inc. “App Store.” *App Store - Support - Apple Developer*, developer.apple.com/support/app-store.

[35] “Apple Announces Mac Transition to Apple Silicon.” *Apple Newsroom*, 1 July 2020, www.apple.com/in/newsroom/2020/06/apple-announces-mac-transition-to-apple-silicon.